Lecture 7: Advanced Object-Oriented Programming I

Instructor: Musard Balliu, `musard@chalmers.se`

`http://www.cse.chalmers.se/~musard`

QUESTIONS?

Plan

- Last time
  1. introduction object-oriented programming
  2. classes and objects

- Today's Plan:

  1. access modifiers

  2. inheritance

  3. interfaces and abstract classes

Access modifiers: public vs private

Java has an additional mechanism for controlling scope, both at the level of types and at the level of variables and methods.

This is done by adding a keyword to the standard declarations. The absence of such a keyword is also significant! (more on this in coming lectures)

Access modifiers for fields and methods

In the following, we consider fields and methods of class `T`.

If a member variable or method declaration is preceded by the keyword `public`, as in

```
public int goals;
public int roll();
```

then that member variable or method can be brought in scope in the usual way from any class in which the type `T` is accessible (i.e., if we have an instance `x` of type `T`, we can use `x.goals` or `x.roll()`).

Access modifiers for fields and methods

If a member variable or method declaration is preceded by the keyword `private`, as in

```
private int goals;
private int roll();
```

then that member variable or method is only in scope in class `T`. `private` variables cannot be brought in scope in any other class.

The `private` modifier

It is a common mistake to confuse a restriction in scope (a static property) with a restriction in object behaviour (a dynamic property).

It might appear that a `private` field of an object cannot be modified by other objects. That, however, is a statement about run-time occurrences, whereas `private` is about static matters. Not surprisingly, the statement is false. An instance of a class has access to all fields of another instance of the same class.

What does the following code do?
Solution.

The `static` modifier

In the general case, each object will have its own copy of the value of a field or the body of a method.

However, if the field or method has been declared `static`, then its value or body will be shared by all objects. In other words, it will be independent of the existence of individual instances.

We have often used such fields and methods, for example:

```
Math.sqrt()
Arrays.sort()
System.out.println()
```

and, of course

```
public static void main(String[] args)
```

The `static` modifier

Because `static` fields and methods are independent of any object, they behave similarly to variables and subroutines in the classical programming languages.

That is why, in the beginning, we prefixed everything with `static`.

Magic is now solved!

Inheritance

Until now, we could only reuse code by packaging it in subroutines/methods.

Classes allow us to reuse code by:

1. creating new datatypes on the basis of existing datatypes, i.e. which have all the fields of an existing datatype without the need for cut-'n-paste;
2. treating instances of these new types as instances of the types they are based on, so they can be used in existing methods.

Inheritance syntax

If `A` is a class, then

```
class B extends A {
    // new fields
    // ...
    // new methods
}
```

will create a new datatype `B`, which, in addition to the new fields and methods will also contain all the fields and methods of `A`. That means that a new method in `B` has in scope all new fields and new methods, and all the fields and methods in `A`.

The fields and methods of `B` are *in a new block*, those of `A` in an enclosing block. That means that, according to the scoping rules, new fields and methods with the same name will shadow the old ones. This is called *overriding*.

Inheritance: “is a”

Instances of type `B` can then be used in any context in which an instance of type `A` can be used. We say, abusively, that an instance of `B` *“is an”* `A`.

In particular, if a variable `a` has been declared to be of type `A`, it can be assigned an expression of type `B`:

```
A a;
a = new B();
```

Example: extending `Player`

Let us first consider the simple `Player` class, with no constructor.

We might want, in addition to name and number of goals, to also take into account the number of games played.

“Old style” we could only do this by adding a field `games` directly to the `Player` class, or by creating an entirely new type with a field of type `Player` and a `games` field.

The disadvantage of the first approach is that we need to recompile all the libraries using the old version of `Player`. The disadvantage of the second approach is that none of the existing functions would work with the completely new type.

By extending the `Player` class, we avoid both disadvantages!

Simple player class
Extended class and exercises
Solutions

Inheritance and constructors

If class `B` extends class `A`, then it inherits the fields and methods of `A`. When a value of type `B` is created, these fields and methods must also be created.

As we have seen, creation of fields and methods is done via constructors, either default, provided by Java, or explicit, provided by us.

For example, when `ibrahimovic = new PlayerG();` above is executed, first the default constructor of `Player` is called, then the fields of `PlayerG` are allocated and initialized to default values.

Inheritance and constructors

However, if we had extended `PlayerC` instead of `Player`, then there would have been no default constructor for the base class, and we would have an error.

Therefore, we either have to implement a constructor with no arguments in class `Player`, or an explicit constructor in class `PlayerC`.

In any constructor of a derived class, the first thing to be executed is a call to the constructor of the base class. This call must always be written as the first statement in the derived class constructor! (Exception: when the base class has a constructor with no arguments, in which case a call to it is automatically inserted by the compiler.)

`super`

The call to this constructor cannot have the form `A();` (e.g. `PlayerC("Ibrahimovic", 5);`) because such a statement will return a new value of type `A`, when in fact we want it to allocate and initialize the fields of the value we are in the process of constructing!

The correct syntax is

```
        super(arg1, ..., argN);
```

The keyword `super` refers to the superclass. In case any fields of the superclass are shadowed in the baseclass, it also allows us to access them. For example, if both the superclass and the subclass contain a method `m()`, then, inside the subclass, a call to `m()` will refer to the local, subclass method. To call the superclass method, we use `super.m()`.

Example

Simple player with constructor
Extended class and exercises
Solutions

The class `Object`

If a class could extend more than one class, ambiguities would arise if fields or methods with the same name existed in the extended classes. In Java, you are only allowed to extend at most one class.

In fact, when creating a new datatype, you are always extending one class. Either explicitly, using `extend`, or implicitly. Java will insert an `extends Object` in all class declaration that do not contain an explicit `extends`.

Thus, an instance of any class “is an” `Object`, and inherits directly or indirectly all the fields and methods of `Object`.

The class `Object`

The class `Object` contains a few fields and methods which are basic for any values of any type. For us, the most important are `toString()` and `equals(Object)`.

`toString()` creates a string representation of an instance of the object. This is, in fact, how such a "general" method as `System.out.println()` can exists: its argument is an instance of `Object` and it uses the `toString()` method of that instance.

Overriding

For most new types, the inherited `toString()` method will not yield good results. For example, if we try to print an instance of our `PlayerC` class, we obtain something like

```
PlayerC@e51510
```

In order to obtain a better result, we must *override* the `toString()` method, by creating a new method with *the exact signature* as `toString()`.

As explained before, since we will have two methods with the same signature, the new one will shadow the outer one.

Overriding `toString()`

As an example, let us override the `toString()` method for the `PlayerC` class.

We can do this by adding a `toString()` method to the `Player` class we already have.

```
public class PlayerC {
    String name;
    int goals;
    //...
    public String toString() {
        return name + " " + goals;
    }

}
```

Overriding `toString()`

Now, when we execute

```
PlayerC ibrahimovic = new PlayerC("Ibrahimovic", 5);
System.out.println(ibrahimovic);
```

we obtain

```
Ibrahimovic 5
```

Extending `PlayerC`

Instead of adding the new `toString()` method to `Player`, we can create a new class, `PrintablePlayer` which extends `PlayerC`.

```
public class PrintablePlayer extends PlayerC {
    public PrintablePlayer(String name, int goals) {
        super(name, goals);
    }
    public String toString() {
        return name + " " + goals;
    }
}
```

Remark: notice the use of `super` in the constructor!
Exercise: what does the code in the `main` method do?

The `PrintablePlayer` class.

Solution.

Interfaces and abstract classes

Now that we can put players in an array, we want to sort the array.

The `Arrays` class in Java provides many sorting routines, but, of course, none that has a `PrintablePlayer` as argument. Instead, they can sort instances of `Object`.

However, in order to sort objects, one has to be able to compare them. But there is no corresponding method in `Object`, since there is no default way of comparing arbitrary objects and deciding which is “better”.

Interfaces and abstract classes

One possibility would have been to create an *abstract class* to represent objects that can be compared. An abstract class is a type for which no instances can be created, because in general it will have “holes”.

The holes are methods which are only declared, not defined. They must be prefixed with the keyword `abstract`. For example:

```
public abstract class Ordered {
    public abstract int compareTo(Object other);
}
```

Interfaces and abstract classes

Subclasses of `Ordered` must then define `compareTo` or be declared abstract themselves.

The sorting method can then be written to work with instances of `Ordered` subclasses of `Ordered`. Note that no instances of `Ordered` as such can exist, but every instance of a subclass of `Ordered` "is an" `Ordered`.

Interfaces and abstract classes

This is not the approach actually used in Java. That is because, in a system which only supports single inheritance, every time the user is required to extend a class, a new level is added to the class hierarchy. This leads to deep hierarchies, which are sometimes difficult to work with.

In order to avoid this, Java provides an additional mechanism which can be used in situations in which a base class only specifies the required signature of methods, but does not implement anything itself. Instead of a class (or an abstract class), one uses an `interface`.

Interfaces are similar to abstract classes, in that they can have no instances. Also, none of the methods may have implementations. Interfaces just collect method declarations.

Since interfaces do not contain any definitions, there is no possible ambiguity in *implementing* them, and a class can implement any number of interfaces simultaneously.

Interfaces and abstract classes

For example, here is the Java interface for objects that can be compared:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

This interface uses *generics*: when implementing the interface, we need to specify the type `T` of objects which we can compare with instances of our class. For example, here is the code which allows us to sort `PrintablePlayers`:

The `PrintablePlayers` with order.

Interfaces and abstract classes

The version we have seen has many flaws. First, we are not really sorting `PrintablePlayers`, but rather `PrintablePlayerOs`. Second, what happens if we want to sort according to the number of goals scored, instead of name? We'd need yet another class, `PrintablePlayerOG`.

The `Arrays` class provides another solution: a sort using a `Comparator`:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
    boolean equals(Object obj);
}
```

Classes which implement `Comparator<T>` define a way in which objects of type `T` can be compared. (By the way, since `Object` defines a default `equalsTo`, only the implementation of `compare` is required).

Interfaces exercises

Using comparators, we can sort `PlayerC` instances directly, without having to create new types. Moreover, we can have one comparator for sorting by name, and another for sorting by goals.

Exercise: implement the two comparators and use them to sort a list of players.

Initial code.
Solution for comparison by name.
Solution for comparison by goals scored.

Homework

Read section 5.5 of David Eck's book.